Representation of the RCS Reference Model Architecture Using an Architectural Description Language

Elena Messina, Christopher Dabrowski, Hui-Min Huang, and John Horst

National Institute of Standards and Technology, Gaithersburg MD 20899, USA
emessina@nist.gov,cdabrowski@nist.gov,horst@nist.gov,hhuang@nist.gov

Abstract. The Real-Time Control System (RCS) Reference Model Architecture provides a well-defined strategy for development of software components for applications in robotics, automated manufacturing, and autonomous vehicles. ADLs are formally defined languages for specification of software system's designs. In this report, we describe the results of an investigation into the use of an ADL to specify RCS software systems, and assess the potential value of ADLs as specification and development tools for RCS domain experts. The report also discusses potential influence of ADLs for commercial software development tools and component-based development.

1 Introduction

Architectural Description Languages (ADLs) are specification languages for rigorously describing and analyzing software system designs. This report provides the results of an investigation into the use of ADLs for formally defining the National Institute of Standards and Technology (NIST) RCS Reference Model Architecture [5].

The RCS Reference Model Architecture provides well-defined guidelines for construction of control software for autonomous real-time systems. We are studying means of formally representing architectures such as RCS in order to facilitate development, understanding, and analysis of complex systems. Communicating RCS in an unambiguous manner was our initial motivation. Beyond that, several potential benefits may accrue. ADLs may provide means of guiding construction of complex systems according to a given reference architecture. This can enhance productivity, reliability, and help ensure conformance to a specified architecture. Analysis tools associated with ADLs may enable developers to study the behaviors and performance of their system design. Although our study focussed on the RCS architecture, we believe that our results are applicable to other architectural models for complex systems.

All significant ADLs were reviewed in a literature search that identified key language features relevant to RCS. Individual ADLs were examined in detail to assess their suitability as specification languages for capturing the structure and function of the RCS Control Node. A detailed, comparative analysis of ADL features was not the scope of this study; readers interested in such an analysis should consult [16]. A single ADL—*Rapide* [15]—was selected to construct a prototype specification of a significant portion of the RCS Intelligent Control Node. The conclusions from out experiment provide a basis for both identifying requirements for ADLs to specify RCS software architectures and components as well as recommending future research directions for ADLs. We believe that our findings can be relevant to the application of ADLs to other complex, real-time architectures.

2 The RCS Reference Architecture

2.1 Overview of RCS Concepts

Developed over the course of two decades at the National Institute of Standards and Technology and elsewhere, RCS has been applied to multiple and diverse systems [4]. RCS application examples include coal mining automation [11], the NBS/NASA Standard Reference Model Architecture for the Space Station Telerobotic Servicer (NASREM) [1], and a control system for a U.S. Postal Service Automated Stamp Distribution Center [26]. Manufacturing applications include the Open Architecture Enhanced Machine Controller [3]and an Inspection Workstation [18].

RCS provides a reference architecture and an engineering methodology to aid designers of complex control systems. Guidelines are provided for decomposition of the problem into a set of control nodes, which are arranged hierarchically. The decomposition into levels of the hierarchy is guided by control theory, taking into account system response times and other factors, such as planning horizons. RCS is focussed primarily on the real-time control domain. It can be further specialized into application-specific versions. 4-D/RCS [5] is one such version, which is aimed at the design and implementation of control systems for intelligent autonomous vehicles for military scout missions. 4-D/RCS has been selected as the software architecture for Department of Defense Demo III eXperimental Unmanned Vehicle (XUV) Program, managed by the Army Research Laboratories [23]. This particular flavor of RCS was studied with respect to ADLs. Figure 1 is a high-level diagram depicting a portion of an RCS hierarchy for an autonomous vehicle. Each node in the hierarchy is built upon the SP-WM-BG-VJ internal elements shown in Figure 2. RCS prescribes a building-block approach to designing and implementing systems. The building blocks are control nodes that contain the same basic elements. The elements, shown in Figure 2, are behavior generation (BG), world modeling (WM), value judgement(VJ), and sensory processing (SP). Associated with WM is a Knowledge Base (KB), which contains longer-term information. Each node receives goals from its superior and, through the orchestration of BG, WM, JV, and SP, generates a finer resolution set of goals for its subordinate nodes. The RCS control node uses an estimated model of the world, generated via SP and WM, to assess its progress with respect to the goals it was given and to make necessary adjustments to its behavior. BG's sub-modules are the job assigner (JA), a set of plan schedulers (SC), a plan

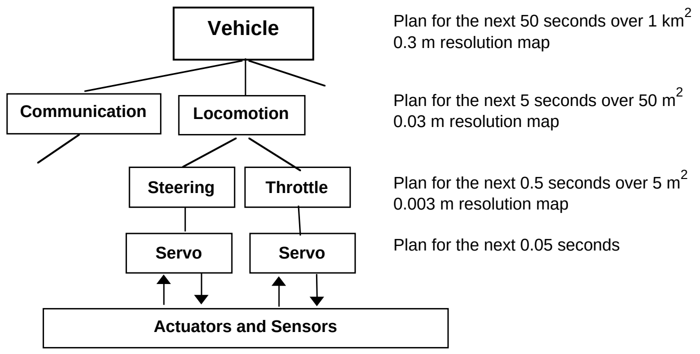
Vehicle
Communication
Locomotion
Steering
Throttle
Servo
Servo
Actuators and Sensors
Plan for the next 50 seconds over 1 km^2
0.3 m resolution map
Plan for the next 5 seconds over 50 m^2
0.03 m resolution map
Plan for the next 0.5 seconds over 5 m^2
0.003 m resolution map
Plan for the next 0.05 seconds

Fig. 1. Example RCS Hierarchy for an Autonomous Scout Vehicle

selector (PS), and a set of executors (EX). One SC and EX exist for each subordinate controlled by a particular RCS node. JA decomposes incoming commands into job assignments for each of its subordinates. Each SC computes a schedule for its given assignment. JA and SC produce tentative plans based on available resources. PS selects from the candidate plans by using WM to simulate the execution of the plans and VJ to evaluate the outcomes of the tentative plans. The corresponding EX executes the selected plan, coordinate actions among subordinates and correcting for errors between the plan and the state of the world estimated by the WM.

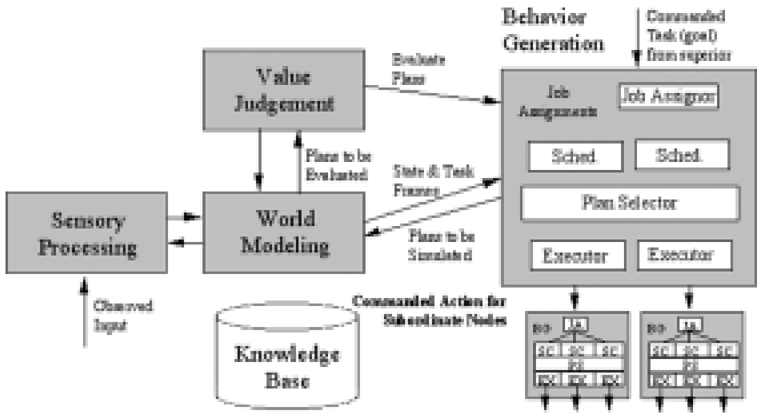

Fig. 2. Model for an RCS Control Node.

3 Architectural Description Languages

3.1 Overview of Architectural Description Languages (ADLs)

Garlan and Perry define a software architecture as consisting of the "structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [9]. An ADL is "a language that provides features for modeling a software system's conceptual architecture" [16].

ADLs provide language constructs for specifying the essential elements of a system's software architecture. A generic set of ADL capabilities has been identified in [10][16][28]. ADLs commonly describe software components by defining their interfaces and behavior in response to externally or internally generated stimuli. An interface definition may include a *signature*, i.e., messages and commands received and sent, as well as constraints on the signature.

Some ADLs support specification of computations performed by a system, referred to as system behavior. Usually, an ADL employs a formally defined descriptive method or underlying *computational* model to provide the necessary semantics. Constraints on behavior are also defined in terms of the computational model. Examples of computational models are Finite State Machines (FSM), Communicating Sequential Processes (CSP) and Partially-Ordered Sets of Events (POSETS). An ADL may allow description of the behavior of component interfaces, component internals, and component connections.

Shaw [22] and Allen [6] provide rules or constraints that place limitations on how components may be connected and what system topologies may be described. One example of an *architectural style* is a top-down hierarchical architecture. Some ADLs allow explicit declaration of architectural styles.

The use of a well defined, rigorous specification language provides a basis for formal analysis of a specification and the verification of software system designs. Some ADLs employ formal proof techniques to determine whether desirable properties, such as internal consistency, hold within a specification. Analysis of ADL specifications may also take place through simulation support tools, which allow the specification to be executed and a result to be computed, thus simulating the computations to be performed by the system being specified.

Gaps in the support provided by object-oriented tools and methodologies [13] further stimulated interest in the potential of ADLs. Object-oriented methods in general are data-centric, providing only for some generic behavior description capabilities. Analysis of the architecture and simulation of the execution of an architecture are not possible in most object-oriented tools. In response to these gaps, the Object Management Group recently issued a Request for Proposals under the title "UML Profile for Scheduling, Performance and Time" [21]. This proposal is aimed at expanding the UML to include support for modeling of time-related paradigms, which are essential for the design and specification of real-time systems.

3.2 The *Rapide* ADL

Rapide [15] is an ADL and supporting tool set developed at Stanford University in the mid-1990s. This ADL was chosen as the primary focus of this study because of its well-developed capability for representing and simulating real-time system designs.

Rapide supports most of the features described above that are common to ADLs. *Rapide* permits definition of a set of component *interface* types, each of which has a signature that includes events generated and received by components of that type and a description of the component's behavior. An interface may also define *constraints* that require dependencies between events, place limitations on the order of events, constrain parameter values, or make other limitations. The internal details of the components themselves—known as *modules*—may also be specified. A module description specifies internal behavior and supporting data structures that allow the module to conform to its *interface*.

The software architecture is formally described by connecting types of events generated in one *interface* specification to events received by another *interface*. A *module* conforming to an *interface* may be decomposed into a sub-architecture consisting of a set of connected component *interfaces*.

Connections between types of events of different *interfaces* and the specification of a component's behavior define causal dependencies of the events. During the simulated execution of a software architecture, these dependencies can be aggregated to form POSETs, or partially ordered set of events.

An event is said to be causally dependent on all events that either directly result in its generation or in the generation of its predecessors. It is independent of all other events. In actual *Rapide* specifications of architectures, very large causal sequences of event types and event constraints can be defined both in interface definitions and as part of connections between interfaces. The causal sequences serve as a basis for "executing" a specification using *Rapide* software support tools to produce simulations.

In *Rapide* the POSET is the basis for automated analysis conducted by an associated toolset. A *Rapide* specification may be defined using the RAPARCH tool, which has a graphical front-end, to specify interfaces and interface connections in a software architecture. When compiled and executed, the *Rapide* specification produces a POSET for the defined architecture. *Rapide* provides a simulation tool called RAPTOR for producing an interactive graphical animation of the execution of the specification in which interfaces and connections are depicted as icons while event icons move between interfaces. The POSET Viewer gives a static picture of a POSET with events and causal arrows between events. Query functions can be used to select interesting subsets of the POSET and provide detailed information. A method is provided for verifying system designs against a more general Reference Model architecture based on comparison of POSETs.

4 The Experiment

In order to help answer the questions about the applicability of ADLs to RCS, the *Rapide* ADL was used to specify a large piece of the RCS Intelligent Control Node. The specification was developed by two of the coauthors: one focusing on the study of ADLs; and the other, a domain expert in design of RCS systems who regularly reviewed the model and guided its evolution. The specification was reviewed and verified by a larger group of experts in RCS. In addition, the use of an ADL to ascertain conformance of individual system designs to the Reference

Model Architecture was examined. A detailed description of the experiment, including the *Rapide* source can be found in [7].

Overview of the Prototype Specification Component interfaces were defined for each 4-D/RCS Intelligent Control Node together with the events handled, sent, and received and applicable constraints for the module. The specification provided the decomposition of the Control Node into its major subcomponents. Behavior Generation was further decomposed into Job_Assigner (JA), a set of Schedulers (SC), a set of Executors (EX), and a Plan Selector (PS). World Modeling (WM) was decomposed into Simulation and Knowledge Base components. The architecture specification included the connections between the interfaces defined for the modules. A sufficient amount of behavior was included to allow the architecture to be simulated using the *Rapide* toolset. The entire specification encompassed more that 1000 lines of *Rapide* code.

Details of Job_Assigner and Scheduler Functions The use of ADLs to specify RCS is illustrated in a sample *Rapide* description of the interaction of two subcomponents of the RCS Behavior Generation Module: The Job_Assigner and a Scheduler (of which there may be several instances). The conceptual design for this representative fragment of the Reference Model functionality is shown graphically in Figure 3. The fragment contains only a subset of the actual events and behavior defined for these components. The specifications of algorithms for computing schedules and selecting plans in underlying modules are omitted from the Reference Model Architecture because they are application-specific.

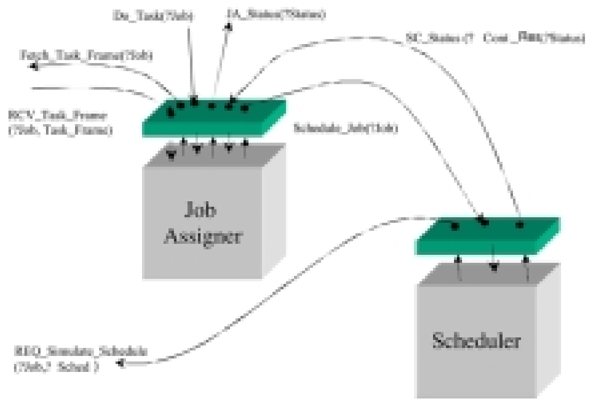
Do_Task(?Job)
JA_Status(?Status)
Fetch_Task_Frame(?Job)
SC_Status (?Cont_Flag(?Status)
RCV_Task_Frame (?Job, Task_Frame)
Schedule_Job(?Job)
Job Assigner
REQ_Simulate_Schedule (?Job,? Sched)
Scheduler

Fig. 3. Job_Assigner and Schedulers in the Behavior Generation Module

Figure 3 shows a *Job_Assigner* component defined as a *Rapide* interface. The *Job_Assigner* interface signature receives a Do_Task event representing an input task in which ?Task is the argument variable for a task name. The *Job_*

Assigner generates a *Fetch_task_frame* event with the job name as an argument that is passed to the *World_Modeling* module. *World_Modeling* returns a task frame data structure containing information necessary to perform the task received by *Job_Assigner* as a *RCV_Task_Frame* event. The underlying module for *Job_ Assigner* decomposes the task frame into job assignments (not shown) for the schedulers. Figure 3 depicts the generation of a *Schedule_Job* event, representing a job assignment to the *Scheduler* interface. The *Scheduler* receives the *Schedule_Job* event. Its underlying module computes a schedule, which is transmitted as an event through the interface outside of *Behavior_Generation* to the *World_Modeling* plan simulator. This is depicted as a plan in Figure 2. Ultimately, the simulated plans are evaluated by *Value Judgement* and returned to the *Plan Selector* in the *Behavior Generation* module (not shown in the example). The *Scheduler* interface is also shown as returning a Status event with a ?Status variable. Values for specific status events would be generated in underlying modules that conform to the interface.

Specification of the Interfaces, Behavior, and Constraints A partial *Rapide* specification of the *Job_Assigner* interface is given below. The *Job_Assigner* is declared to be of type Interface. The signatures for the events received by, and sent from this interface are provided including variable arguments and their types.

```
TYPE Job_Assigner_Interface IS INTERFACE;
ACTION
        IN
                Do_Task (Task : Task_Command_Frame),
                RCV_task_frame (Task : Task_Command_Frame; TF : Task_Frame),
                SC_Status (CR : Controlled_Resources; ST : String);
        OUT
                Schedule_Job (Job : Task_Command_Frame),
                Fetch_task_frame (Task : Task_Command_Frame),
                Decompose_task_frame (TF : Task_Frame),
                JA_Status (?status);
BEHAVIOR
        (?Task : Task_Command_Frame)
         Do_Task (?Task) ||> Fetch_task_frame (?Task);
        (?Task : Task_Command_Frame; ?TF : Task_Frame)
         RCV_task_frame (?Task, ?TF) ||> Decompose_task_frame(?TF);;
END;
```

A portion of the behavior depicted in Figure 3 is also specified. The receipt of a Do_Task command to perform a task triggers a request for a task frame containing essential information needed to perform the task. A causal connection is defined between these two events. The receipt of a RCV_task_frame command results in a Decompose_task_frame in which (?TF) denotes the variable placeholder for the task frame which is transferred. The Schedule_Job and Status events are generated through the interface by underlying conforming modules which also instantiate the necessary arguments. These are omitted from this portion of the specification example.

The specification of the Job_Assigner is supplemented by the declaration of constraints shown below.

```
CONSTRAINT
– (C1) Do not allow causally independent Do_Task and Schedule_Job events
NEVER (?Task : String; ?Job : String)
            Do_Task (?Task) || Schedule_Job (?Job);
– (C2) Do not allow causally independent Do_Task and Status Message events
NEVER (?Task : String; ?status : String)
            Do_Task (?Task) || JA_Status (?status);
```

Constraint C1 prohibits the independence of Do_Task and Schedule_Job events, while constraint C2 prohibits independence of Do_Task and Status_Events. These constraints require that that these events *must always be* related in a causal sequence.

Specification of the Architecture The specification of the portion of the Behavior Generation architecture from Figure 3 is given below. This specification shows the connection of the events between the Job_Assigner, an array of Schedulers and the Plan Selector.

```
ARCHITECTURE BG_Module_Arch () ...
IS
      JA : Job_Assigner_Interface IS Job_Assigner_Module();
      SC : array [integer] of Scheduler_Interface IS
            (1..$Num_Controlled_Resources,...)
      PS : Plan_Selector_Interface IS Plan_Selector_Module();...
CONNECT
      (?Job : Task_Command_Frame)
      JA.Schedule_Job(?Job) ||> SC i.RCV_Schedule_Job(?Job);
      (?CR : Controlled_Resources; ?ST : String)
      SC[ i].SC_Status (?CR, ?ST) ||> JA.SC_Status (?CR, ?ST);
      ...
      (?CR : Controlled_Resources; ?Job : Task_Command_Frame;
      ?Sched : Schedule; ?ST :string)
      PS.SND_PS_Status (?CR, ?Job, ?Sched, ?ST) ||>
            SC[i].RCV_PS_Status (?Job, ?Sched, ?ST);
```

Note that each of these components is declared as an instance of one of the types defined above. This is followed by explicit connections between OUT events in the interface of one component and IN events in another interface via the CONNECT keyword. The *Rapide* symbol "||>" is used to indicate a causal connection between these events.

Execution of the RCS Intelligent Control Node Architecture The declaration of causal connections between events in *Rapide interfaces* and in the declaration of the architectures defines a causal sequence of events. The execution of this architecture produces a POSET, which can be used to analyze sequences of events and causality. A portion of the POSET generated by the execution of the RCS control node is shown in Figure 4, which omits intervening events not described in the partial specifications given above. The figure shows the causal connection between the Do_Task event and a Fetch_Task_Frame event that retrieves information necessary to initiate scheduling activity. When the Job_Assigner receives the Task_Frame, this triggers the Decompose_Task_Frame event followed by the Schedule_Job event that is forwarded to a set of Schedulers. POSETs such as the one shown were useful for analyzing sequences of events and communicating behavior of the architecture.

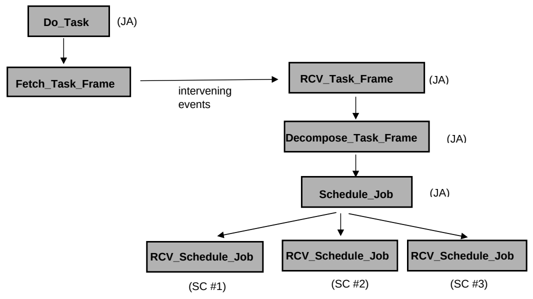

Fig. 4. Event trace of *Rapide* Reference Model Specification

Verification of Individual System Designs Against the Reference Model Architecture *Rapide* provides a capability for verifying that the behavior of a system design, or *concrete* architecture, conforms to that of a more *abstract* architecture, such as the RCS Reference Model Architecture. This is accomplished by first declaring a set of constraints in the abstract architecture and then declaring an equivalence, or mapping, of events from the *concrete* to the *abstract* architecture. The *abstract* architecture is then executed with the "mapped" events of the *concrete* architecture replacing events originally defined in the *abstract* architecture to create a POSET event trace similar to Figure 4. Conformance to constraints of the *abstract* architecture is tested. If constraints are violated, error messages appear in the POSET as events, indicating that the concrete architecture is non-conformant. The conformance verification feature was exercised and found to be useful. One of the challenges facing RCS developers is ensuring that their systems comply with the RCS reference model architecture.

5 Conclusions and Recommendations

5.1 Specifying and Analyzing RCS

Based on informal review by RCS experts, the Intelligent Control Node specification was successful in capturing and representing major RCS architectural concepts. To date, it is the most rigorous representation of the reference model architecture. However, the specification had to be simplified and modified to allow the application of specific RCS keywords, and supplemented by the use of graphical support. The simulated execution of the Control Node Architecture reinforced the specification and proved to be a valuable aid in communicating the architecture by enabling reviewers to visualize the topology and high-level execution of the Intelligent Control Node.

The structure and behavior of the RCS Reference Model Architecture were captured by *Rapide*. Aided by the simplification of the specification and the use of graphics, the Control Node module Interfaces and signatures were clearly defined. Module connections, even though not definable in *Rapide* as explicit types–or first-class objects–were also easily communicated. The successful representation of the RCS Control Node hierarchy indicates that representation of other parts of the multi-level architecture described in Section 2 should be possible.

The ability to create a precise, communicable specification of the RCS Reference Model Architecture led to potential improvements to the architecture itself. Two possible changes to the Architecture as described in [5] were identified, one of which will be described. In the model described in Section 4, Job_Assigner applies a Fetch_Task_Frame operation to retrieve the task knowledge necessary for task decomposition. Although this operation is not explicitly stated in the RCS Reference Model, we found it consistent with the usage of task frames and found it effective in our experiment. Therefore, this operation may be proposed as one of the accepted Job_Assigner functions in its specification. This illustrates the potential of ADLs as practical tools for development of the Reference Model Architecture and software designs in general.

The use of an ADL to verify the behavior of an application system design against the Reference Model Architecture was demonstrated as a proof-of-concept in the Control Node prototype. However, RCS domain experts maintain that verification of the system topology is at least equally important for the Reference Model Architecture. This form of verification involves showing that the application system contains the same basic structure including components, event connections, and data structures as the Reference Model. As a result, two kinds of verification are important from the standpoint of RCS. The first is verification to the structure of the Reference Architecture including existence of specific components, events, and control flows. The second is verification of behavior, including behavior within components and behavior across component connections and an entire architecture. Verification of behavior is the focus of *Rapide*.

Further research is necessary to define techniques for demonstrating consistency with system topology. Work in extending *Rapide*'s POSET model to verification of system structure has been reported in [27]. In SADL [19], Moriconi describes a general approach, called *architectural refinement*, that utilizes theorem proving techniques. In this approach, proofs are constructed to show that in the case when a more general or abstract architecture is applied to produce a more detailed design, that any system that correctly implements the more detailed design also correctly implements the abstract architecture. Refinement is used to demonstrate correctness with respect to the connectivity of events between modules at different levels of abstraction; and this approach may be applicable to the problem of verifying application system designs.

5.2 Appropriate Abstractions for RCS Architectures

Owing to evidence in biological systems and theory of control science, RCS prescribes rules for decomposing the control hierarchy for a system. In his "Outline for a Theory of Intelligence," [2], Albus proposed that:

"In a hierarchically structured, goal-driven sensory interactive, intelligent control system architecture, control bandwidth decreases about an order of magnitude at each higher level, perceptual resolution of spatial and temporal patterns decreases about an order of magnitude at each higher level, goals expand in scope and planning horizons expand in space and time about an order of magnitude at each higher level, and models of the world and memories of events decrease in resolution and expand in spatial and temporal range by about an order of magnitude at each higher level."

These English language rules must be encoded into ADLs in order to represent fully the semantics of an RCS system. Temporal scales and spatial extents relative to other levels of the hierarchy must be represented and validated. While existing ADLs can meet some of these requirements, further work on ADLs adding methods to define these measures and to express constraints among them is needed to allow specifications such as those quoted to be stated and applied.

The syntactic description was simplified and altered to conform to the descriptive forms familiar to RCS experts. RCS experts found specifications much easier to understand when RCS terminology was used. As an example of this approach, instead of declaring an RCS module such as SCHEDULER as a *component* or *interface type*, it should be possible to introduce a higher-level language type called *RCS_Module* in a specification that could serve as a "meta type" for the definition of interface types that are specific to RCS such as SCHEDULER. As a long-term goal, ADLs should allow specifications to be stated at a sufficiently high level of abstraction for non-computer scientists so that they are easier to understand than a program written in C++. This argues for the development of either a flexible ADL with an extensible syntax that can be specialized for RCS or a domain-specific ADL that utilizes RCS terminology.

To facilitate communication of RCS system behavior, an ADL must provide an effective means for abstractly specifying algorithms, component behavior, and performance. While some ADLs may allow representation of all or most of the behavior needed for RCS, this requirement may lead to defining additional language constructs to more directly represent specific RCS behavior. It may also require additional facilities for guiding developers in generating their component specifications, through for example, templates that they can fill in, as proposed in [12] and [17]. As with system structure, such capabilities would allow ADLs to specify essential aspects of behavior at a higher level of abstraction than for programming languages. These capabilities could be part of a domain-specific ADL with a syntax that is customized for RCS systems.

5.3 General Software Development Support

It is often important to be able to divide the processing into atomic processing components that can be executed serially or in parallel, which will facilitate process cessation and make it deterministic. Therefore, it is important that an ADL be able to specify processing characteristics such as process modularization, parallel and serial execution. Serial and parallel processing capabilities are provided by some ADLs, including *Rapide*.

It is desirable to allow capturing performance statistics, such as timing, states, and errors. These would be useful in system diagnostics and maintenance. It should be noted that *Rapide* does provide the capability to capture time-related data which could not be exercised in this study due to resource limitations.

For designing real-time systems, an ADL should define notions of duration in time of processes, mixed asynchronous and synchronous processing, spatial scope of a process or set of processes, algorithm and component complexity, and determinism in execution.

One of the benefits of rigorously specifying RCS designs is that it is possible to check the completeness and internal consistency of the reference model architecture before it is used as a basis for developing individual system designs. By providing a basis for formalized, or at least rigorous specification, most ADL products surveyed also provide a basis for development of automated analysis capabilities. In the case of *Rapide*, analysis is based on simulation of the execution of a system architecture and analysis of POSET traces. This proved to be valuable for visualizing, understanding, and verifying system behavior.

Other ADLs take different approaches using automated tools for analysis of specifications based on formal methods approaches. *SADL* [20] uses w-logic, a weak second-order logic, as a basis for proving the correctness of mappings between architectures at different levels of abstraction. *Wright* [6] uses First-Order Logic to specify constraints and a Communicating Sequential Processes (CSP) computational model to specify behavior of components and connections, providing a basis for a set of automated checks on specification consistency and completeness. Examples from *Wright* are checks that determine the existence of a deadlock condition within the specification of the behavior of an architecture and checks to determine compatibility between connections and components.

5.4 Transfer of ADL Concepts into Real-Time Development Tools

Presently, there are a number of public domain and commercially available software support tools for design and simulation of real-time software systems. These tools have well-developed facilities for designing and implementing individual software systems. However, they do not typically provide any guidance to users about how to structure their system or make other design decisions. ADLs introduce notions of software architecture that could potentially provide additional structure in order to improve the capabilities of these tools. Users or enterprises could set preferences in term of which architecture or architectural style is to be used in developing systems. The tools would then either guide designers as the system is being developed or could flag situations where the architecture or style are violated. Further effort is necessary to explore the potential of infusing ADL concepts into real-time development support tools. This avenue could provide the benefits of ADLs to end users while shielding them from having to learn a new language and concepts. The real-time development tools would guide users in constructing systems per rules for a prescribed architecture through their graphical user interfaces. The users would not be burdened with the underlying mechanics of the ADL specification. In addition to design, analysis and simulation capabilities from the ADLs could be incorporated into the tools. The tools could generate executable or source code. This would automatically assure traceability from the desired architecture through to the executable code. Eventually, tools using ADLs could support highly automated composition of real-time systems from existing or tailorable components.

5.5 ADLs and Component-Based Software Reuse

There is potentially a strong relationship between ADLs and component-based software reuse. ADLs go beyond providing just the signature specification for a component or subsystem. They allow developers to see the big picture, where their particular pieces fit in, and how the pieces are expected to behave or interact with the rest of the system. Simulation of components provides additional benefits not available in typical notations or descriptions of software components.

ADLs could be extended to support reuse with additions of specific language features based on reuse concepts from the literature on domain engineering [14] [24] [25], thus providing a basis for automation of software development. Domain engineering is the process of developing reusable software for a family of systems with similar requirements. An architecture specification may identify optional components, parameterizable components, or even entire subarchitectures that can be varied. Guidelines would be used by developers with the aid of support tools to select options and customize the specification for particular applications. This concept could be further extended by the use of software support tools that assist developers in selecting and modifying system designs and components. The resulting system specifications potentially could be automatically composed and generated using the support tools. An example of such a system for automated generation of system requirements is provided in [8].

6 Summary

This report has provided the results of an investigation into the use of architectural description languages to represent the RCS Reference Model Architecture and RCS software components. ADLs have the capabilities to represent RCS and to be useful tools for further developing RCS. However, several areas of research are suggested in order to make ADLs more effective tools for RCS software specifications. Transfer of ADL concepts into existing real-time software development tools is another important direction to pursue. It is the hope of the authors that this work provides a contribution towards both the development of ADLs as tools for software component technology and the formalization of the RCS Reference Model Architecture.

Acknowledgements The authors wish to thank John Kenney, David Luckham, and other members of the Stanford University *Rapide* Project for their generous assistance with the *Rapide* ADL and software support tools. Thanks is also provided to NIST staff members who reviewed this paper and the RCS prototype specification and provided critical commentary.

References

1. Albus, J.S., Lumia, R., Fiala, J., and Wavering, A. 1989. NASREM - The NASA/NBS Standard Reference Model for Telerobot Control System Architecture. Proc. of the 20th International Symposium on Industrial Robots, Tokyo, Japan.
2. Albus, J. S. 1991. "Outline for a Theory of Intelligence. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 3:473-509.
3. Albus, J.S., Lumia, R. 1994. The Enhanced Machine Controller (EMC): An Open Architecture Controller for Machine Tools. Journal of Manufacturing Review, Vol. 7, No. 3, pgs. 278-280.
4. Albus, J. S. 1995. The NIST Real-time Control System (RCS): An Application Survey. Proc. of the AAAI 1995 Spring Symposium Series, Stanford University, Menlo Park, CA.
5. Albus, J. S. 1997. RCS: A Reference Model Architecture Demo III. National Institute of Standards and Technology, Gaithersburg, MD, NISTIR 5994.
6. Allen, R. 1997. A Formal Approach to Software Architecture. PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, Technical Report Number: CMU-CS-97-144.
7. Dabrowski, C., Huang, H., Messina, E., Horst, J., 1999. Using Architectural Description Languages to Formalize the NIST 4-D/RCS Reference Model Architecture. National Institute of Standards and Technology Draft NISTIR.
8. Dabrowski, C. and Watkins, C. 1994. A Domain Analysis of the Alarm Surveillance Domain. National Institute of Standards and Technology, Gaithersburg, MD, NISTIR 5494.
9. Garlan, D., and Perry, D. 1995. Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 269-274.
10. Garlan, D., and Shaw, M. 1994. Characteristics of Higher-Level Languages for Software Architecture. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, CMU/SEI-94-TR-23.
11. Horst, J. A. 1993. Coal Extraction Using RCS. Proc. of the 8th IEEE International Symposium on Intelligent Control, Chicago, IL, pp. 207-212.
12. Horst, J. A., Messina, E., Kramer, T., Huang, H. M. 1997. Precise Definition of Software Component Specifications. Proc. of the 7th Symposium on Computer-Aided Control System Design (CACSD '97), Gent, Belgium, pp.145-150.

13. Huang, H. and Messina, E. 1996. NIST-RCS and Object-Oriented Methodologies of Software Engineering: A Conceptual Comparison. Proc. of the Intelligent Systems: A Semiotic Perspective Conference, Vol. 2: Applied Semiotics. Gaithersburg, MD, pp. 109-115.
14. Kang, K., Cohen S. , Hess J., Novak W., and Peterson S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, CMU/SEI-90-TR-21.
15. Luckham, D. 1996. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. Stanford University, Palo Alto, CA. CSL-TR-96-705.
16. Medvidovic, N. and Taylor R. 1999. Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in IEEE Transactions on Software Engineering.
17. Messina, E., Horst, J., Kramer, T., Huang, H. Michaloski, J. 1999. Component Specifications for Robotics Integration. Autonomous Robots Journal, Volume 6, No. 3, pp. 247-264.
18. Messina, E., Horst, J., Kramer, T., Huang, H., Tsai, T., Amatucci, E. A Knowledge-Based Inspection Workstion. Proc. of the IEEE International Conference on Information, Intelligence, and Systems. Bethesda, MD. November, 1999.
19. Moriconi, M., Qian, X. and Riemenschneider, R. 1995. "Correct Architecture Refinement. IEEE Transactions on Software Engineering, Volume 21, Number 4, pp.356-372.
20. Moriconi, M and Riemenschneider, R. 1997. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Stanford Research Institute, Palo Alto, CA, TR SRI-CSL-97-01.
21. OMG. 1999. RFP: UML Profile for Scheduling Performance, and Time Object Management Group Document ad/99-03-13. Object Management Group, Framingham, MA. http://www.omg.org.
22. Shaw, M. 1994. Comparing Architectural Design Styles. IEEE Software, November, 1994, pp. 27-41.
23. Shoemaker, C. M. and Bornstein, J. A. 1998. Overview of the Demo III UGV program. Proc. of the SPIE Robotic and Semi-Robotic Ground Vehicle Technology , Vol. 3366, pp.202-211.
24. SPC 1992. Domain Engineering Guidebook, Software Productivity Consortium. Herndon, VA. SPC-92019-CMC, Version 01.00.03.
25. STARS. 1993. Organizational Domain Modeling, Volume I - Conceptual Foundations, Process And Workproduct Description, Informal Technical Report for the Software Technology for Adaptable, Reliable Systems (STARS), Report Number STARS-UC-05156/024/00.
26. USPS. 1991. Stamp Distribution Network, Advanced Technology & Research Corporation, Burtonsville, MD. USPS Contract Number 104230-91-C-3127 Final Report.
27. Vera, J., Perrochon, L., Luckham, D. 1998. Event-Based Execution Architectures for Dynamic Software Systems. Proc. TC2 First Working IFIP Conference on Software Architecture (WICSA1). San Antonio, Texas, USA. Kluwer. pp. 303-317.
28. Vestal, S. 1993. A Cursory Overview and Comparison of Four Architecture Description Languages. Honeywell Technology Center, February 1993.